PCSG: Pattern-Coverage Snippet Generation for RDF Datasets

Xiaxia Wang[1], Gong Cheng[1], Tengteng Lin[1], Jing Xu[1], Jeff Z. Pan[2], Evgeny Kharlamov[3,4], and Yuzhong Qu[1]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, China
{xxwang,tengtenglin,jingxu}@smail.nju.edu.cn, {gcheng,yzqu}@nju.edu.cn
[2] School of Informatics, University of Edinburgh, UK
https://knowledge-representation.org/j.z.pan/
[3] Bosch Center for Artificial Intelligence, Robert Bosch GmbH, Germany
[4] Department of Informatics, University of Oslo, Norway
evgeny.kharlamov@de.bosch.com

Abstract. For reusing an RDF dataset, understanding its content is a prerequisite. To support the comprehension of its large and complex structure, existing methods mainly generate an abridged version of an RDF dataset by extracting representative data patterns as a summary. As a complement, recent attempts extract a representative subset of concrete data as a snippet. We extend this line of research by injecting the strength of summary into snippet. We propose to generate a pattern-coverage snippet that best exemplifies the patterns of entity descriptions and links in an RDF dataset. Our approach incorporates formulations of group Steiner tree and set cover problems to generate compact snippets. This extensible approach is also capable of modeling query relevance to be used with dataset search. Experiments on thousands of real RDF datasets demonstrate the effectiveness and practicability of our approach.

Keywords: RDF data · Snippet · Data pattern · Dataset search.

1 Introduction

We have witnessed increasingly many RDF datasets published on the Semantic Web, but understanding the content of a large RDF dataset is still a challenge. Fruitful efforts have been made to compute and present an abridged version of an RDF dataset by extracting representative data patterns to form a *summary* [2]. Summaries are typically composed of schema-level elements, i.e., classes and properties [14, 7, 24, 28, 29]. Complementary to the aggregate nature of summaries, a recent line of research extracts a representative subset of instance-level triples to form a compact *snippet* exemplifying concrete data in an RDF dataset [6, 15]. We follow this trend to generate snippets that can be incorporated into RDF dataset search engines and profiling tools used by human users.

Research Questions. Existing methods [6, 15] generate a snippet by extracting a compact connected RDF subgraph that covers the most important classes and properties in an RDF dataset. There are three limitations of these methods, which pose three research questions (RQs) accordingly. Firstly, these methods aim at covering representative schema-level elements, but they are not powerful enough to attend to combinations of these elements in entity descriptions, i.e., patterns which have been extensively studied in summary generation [14, 7, 24, 28, 29]. **RQ1:** How can we generate compact pattern-coverage snippets for RDF datasets? Secondly, while a snippet being a connected RDF subgraph might benefit users' understanding, if the original RDF dataset comprises multiple components, however, connectivity will force existing methods [6, 15] to extract a snippet from only one component but ignore the content of all other components. Alternatively, if we extract a sub-snippet from each component and merge all sub-snippets, we will be likely to suffer from redundancy and inefficiency. **RQ2:** How can we jointly consider all components to generate a compact snippet? Thirdly, dataset search [3] is a major downstream application of snippets, and it is often triggered by a query which cannot be exploited by the above methods [6, 15]. Query-dependent snippets may help users better determine the relevance of retrieved RDF datasets. **RQ3:** How can we extend snippet generation to be biased toward a given query?

Research Contributions. To answer the above RQs, we inject the strength of summarization (i.e., pattern) into snippet generation and combine the two lines of research. We propose to generate compact *pattern-coverage snippets* for RDF datasets. We answer the three RQs with the following research contributions.

– **For RQ1:** We present an algorithm `Basic` for generating a compact snippet covering all the patterns of entity descriptions and links in an RDF dataset. `Basic` achieves compactness by solving a group Steiner tree problem.
– **For RQ2:** Using `Basic` as a subroutine, we present an algorithm `PCSG` which handles disconnectivity by generating a compact pattern-coverage snippet that merges the smallest number of sub-snippets extracted from different components. `PCSG` achieves compactness by solving a set cover problem.
– **For RQ3:** We present an algorithm `QPCSG` which extends `PCSG` to generate a query-biased pattern-coverage snippet. `QPCSG` covers each query keyword as a pseudo-pattern of its matching entity descriptions and links.

Outline. The remainder of the paper is organized as follows. We discuss related work in Section 2. We describe `Basic` in Section 3. We describe its extensions `PCSG` and `QPCSG` in Section 4, and we evaluate these algorithms in Section 5 and Section 6, respectively. We empirically compare snippet with summary in Section 7. We conclude the paper with future work in Section 8.

2 Related Work

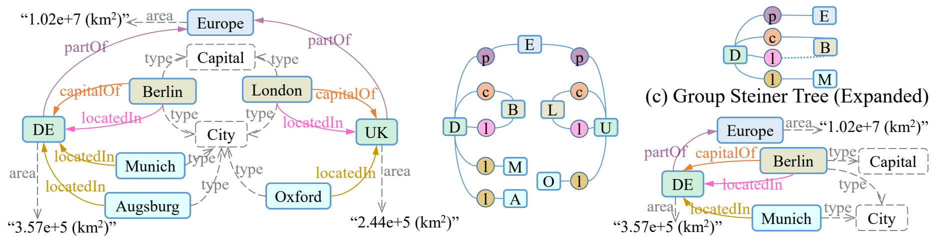
Given a piece of RDF data [18], a snippet is a subset of triples. Various kinds of snippets have been generated to facilitate different downstream tasks.

RDF Dataset Snippet. To compactly exemplify the content of a large RDF dataset, IlluSnip [6] generates a snippet by formulating a maximum-weight-and-coverage connected graph problem. It aims at extracting an optimum subset of k triples represented as a connected RDF graph that covers the most frequent classes, properties, and the most central entities in the RDF dataset. An approximation algorithm is designed for this NP-hard problem. In [15], a more scalable anytime version of IlluSnip is presented and it can generate snippets for RDF datasets accessible via SPARQL endpoints. Different from IlluSnip, KSD [6] formulates a weighted maximum coverage problem where it removes the constraint on connectivity. Its objective of optimization further aims at covering the most keywords in a keyword query so that it is suitable for RDF dataset search engines. To evaluate these snippets, in [26] a set of metrics are defined to measure how many important classes, properties, entities, and keywords are covered in a snippet. Compared with IlluSnip and KSD, while our approach also aims at covering schema-level elements, we focus on patterns of entity descriptions and links which are *combinations* of classes and properties. Patterns can provide a "higher-order" preview of data than *separate* classes and properties.

RDF Dataset Sample. To efficiently answer SPARQL queries over an RDF dataset, SampLD [20] creates a sample to replace the original RDF dataset. It extracts central triples from the RDF dataset as they are considered to frequently appear in the answers to common queries. GLIMPSE [21] has a similar goal but its ranking of triples is personalized, i.e., biased toward a user's query history. In [12], a sample is created to capture the structural and statistical features of an RDF dataset to benefit query plan optimization. Compared with our dataset snippets which are generated to be read by *human users*, dataset samples are created to be used by *machines* in SPARQL query processing. The two problems and their solutions are fundamentally different.

Entity Summary. The research on entity summarization aims at generating a representative snippet called an entity summary for RDF data that describes *a specified entity* to show its main features [16]. Methods addressing this problem compute a ranking of triples but they cannot apply to an RDF dataset containing *many and various entities* studied in our work.

RDF Dataset Summary. A summary of an RDF dataset usually refers to a set of patterns in the data [2]. Patterns are combinations of classes and properties [14, 7, 24, 28, 29], or more complex path-based patterns [1, 23]. Snippet and summary provide *complementary views* of an RDF dataset: snippets containing representative *instance-level* triples; summaries comprising representative *schema-level* patterns. They are both important features of a dataset profile [8]. Our approach *combines their strengths* by generating a pattern-coverage snippet. Different from our focus, there are also dataset summaries that can be used for optimizing distributed query answering [10, 11] or vocabulary reduction [25].



(a) Example RDF Dataset (b) Entity-Link Graph (ELG) (c) Group Steiner Tree (Expanded) (d) Pattern-Coverage Snippet

Fig. 1: An RDF dataset and its pattern-coverage snippet generated by `Basic`.

3 Snippet Generation: A Basic Approach

3.1 Problem Formulation

RDF data is a set of subject-predicate-object triples and can be represented as an RDF graph. An example RDF dataset is shown in Fig. 1(a).

Snippet. Given an RDF dataset D which is a set of triples, a *snippet* S of D is a subset of triples represented as a connected RDF graph [6, 15]. Connectivity indicates that S describes a set of interlinked entities and exhibits cohesion which is beneficial to users' understanding [6]. In this section we follow this definition.

However, if D itself is represented as a disconnected RDF graph, a snippet defined as above can only be generated from one component and will have to ignore the data in other components. To overcome this limitation, while in this section we assume D is represented as a connected RDF graph, in Section 4.1 we will cope with disconnectivity in our full approach.

Pattern. Given a set of triples T, an instance-level entity e is described by a subset of triples where e is the subject or the object. The schema-level elements in these triples form the *entity description pattern* (EDP) of e, consisting of sets of classes (`C`), forward properties (`FP`), and backward properties (`BP`):

$$\begin{aligned} \texttt{edp}(e,T) &= \langle \texttt{C}(e,T),\ \texttt{FP}(e,T),\ \texttt{BP}(e,T)\rangle\,, \\ \texttt{C}(e,T) &= \{c : \exists\langle e,\ \texttt{rdf:type},\ c\rangle \in T\}\,, \\ \texttt{FP}(e,T) &= \{p : \exists\langle e,\ p,\ o\rangle \in T\} \setminus \{\texttt{rdf:type}\}\,, \\ \texttt{BP}(e,T) &= \{p : \exists\langle s,\ p,\ e\rangle \in T\}\,. \end{aligned} \tag{1}$$

A triple where the object is an entity is of particular interest as it represents a link between two entities. The predicate and the EDPs of the two entities in such a triple $\langle e_i,\ p,\ e_j\rangle$ form the *link pattern* (LP) of this triple:

$$\texttt{lp}(\langle e_i,\ p,\ e_j\rangle, T) = \langle \texttt{edp}(e_i,T),\ p,\ \texttt{edp}(e_j,T)\rangle\,. \tag{2}$$

For example, given T comprising all the triples in Fig. 1(a), we use different colors to show entities and links in Fig. 1 having different patterns such as

$$\begin{aligned} &\texttt{edp}(\texttt{Berlin},T) = \texttt{edp}(\texttt{London},T) = p_1 = \langle\{\texttt{Capital},\texttt{City}\},\{\texttt{capitalOf},\texttt{locatedIn}\},\emptyset\rangle \\ &\texttt{edp}(\texttt{DE},T) = \texttt{edp}(\texttt{UK},T) = p_2 = \langle\emptyset,\{\texttt{partOf},\texttt{area}\},\{\texttt{capitalOf},\texttt{locatedIn}\}\rangle \\ &\texttt{lp}(\langle\texttt{Berlin},\texttt{locatedIn},\texttt{DE}\rangle,T) = \texttt{lp}(\langle\texttt{London},\texttt{locatedIn},\texttt{UK}\rangle,T) = \langle p_1,\texttt{locatedIn},p_2\rangle\,. \end{aligned}$$

By iterating over all entities and links in T, we obtain the set of all EDPs and the set of all LPs in T, denoted by $\texttt{EDP}(T)$ and $\texttt{LP}(T)$, respectively.

Pattern-Coverage Snippet. Given an RDF dataset D, a *pattern-coverage snippet* S of D is a snippet that covers all the EDPs and LPs in D:

$$\texttt{EDP}(D) = \texttt{EDP}(S) \quad \text{and} \quad \texttt{LP}(D) = \texttt{LP}(S)\,. \tag{3}$$

For example, Fig. 1(d) shows a pattern-coverage snippet of the RDF dataset in Fig. 1(a). Observe that S may not be unique. For example, $S = D$ is a trivial pattern-coverage snippet. We aim at finding a compact S of the *smallest size* in terms of the number of triples. We refer to this optimization problem as the *pattern-coverage snippet problem* (PCSP). We will present a solution to PCSP in Section 3.2.

Note that if the heterogeneity of D is very high containing many different EDPs and LPs, S covering all patterns can hardly be very compact. In Section 4.2 we will extend our approach to cope with high heterogeneity.

3.2 Algorithm `Basic`

We solve PCSP by Algorithm `Basic`. Its three steps are outlined in Algorithm 1. We illustrate the output of each step in Fig. 1(b), Fig. 1(c), and Fig. 1(d).

Step 1. We firstly represent an RDF dataset D as an undirected graph where nodes and edges represent entities and entity links in D, respectively. Each node is labeled with its EDP, and each edge is labeled with its LP. Then we convert labeled edges into labeled nodes by subdividing each edge. The subdivision is referred to as the *entity-link graph* representation of D, denoted by $\texttt{ELG}(D)$, as illustrated by Fig. 1(b) where different labels are represented by different colors.

Step 2. Observe that PCSP essentially looks for a smallest connected subgraph of $\texttt{ELG}(D)$ whose node labels cover $\texttt{EDP}(D)$ and $\texttt{LP}(D)$. It would be straightforward to reduce PCSP to an unweighted version of the well-known *group Steiner tree problem* (GSTP): all nodes having the same label form a group. GSTP requires finding a smallest tree that connects at least one node from each group and hence it covers all distinct labels. GSTP is NP-hard, and we solve it using KeyKG+ [22], a state-of-the-art approximation algorithm for GSTP. Note that for each leaf in the computed tree representing an entity link, we expand the tree to contain both entities it links, as illustrated by the dotted edge in Fig. 1(c).

Step 3. From the computed subgraph of $\texttt{ELG}(D)$ we derive a pattern-coverage snippet S as follows. For each node in the subgraph representing an entity e, from the triples describing e in D: we choose all triples describing e's classes, and for each property in $\texttt{edp}(e, D)$ we choose an arbitrary triple describing e using this property. For each node in the subgraph representing an entity link: we choose its corresponding triple from D. All the chosen triples form S, as illustrated by Fig. 1(d).

```
Algorithm 1: Basic
   Input: An RDF dataset $D$.
   Output: A pattern-coverage snippet $S$.
1  Construct ELG($D$);
2  Compute a group Steiner tree in ELG($D$);
3  Derive $S$ from the computed subgraph;
4  return $S$;
```

```
Algorithm 2: PCSG
   Input: An RDF dataset $D$.
   Output: A pattern-coverage snippet $S$.
1  $\mathcal{D} \leftarrow$ Components($D$);
2  $P \leftarrow$ EDP($D$) $\cup$ LP($D$);
3  $S \leftarrow \emptyset$;
4  while $P \neq \emptyset$ do
5      $D_i \leftarrow \arg\max_{D_j \in \mathcal{D}} |(\mathtt{EDP}(D_j) \cup \mathtt{LP}(D_j)) \cap P|$;
6      $S \leftarrow S \cup$ Basic ($D_i$);
7      $P \leftarrow P \setminus (\mathtt{EDP}(D_i) \cup \mathtt{LP}(D_i))$;
8  return $S$;
```

Time Complexity. The run-time of `Basic` is dominated by KeyKG+ in Step 2. KeyKG+ runs in $O(n^2 g + ng^3)$ time [22], where $n \leq 3|D|$ is the number of nodes in $\mathtt{ELG}(D)$, and $g = |\mathtt{EDP}(D) \cup \mathtt{LP}(D)|$ is the number of groups. Thanks to the efficiency of KeyKG+ [22], `Basic` is also efficient and practical as we will see in the experimental results in Sections 5 and 6.

4 Snippet Generation: Extended Approaches

In this section, we extend `Basic` to accommodate more general settings.

4.1 Extension to Disconnectivity: Algorithm `PCSG`

`Basic` assumes the connectivity of D. We use it as a subroutine to be called by our main algorithm `PCSG` which is extended to handle disconnectivity as follows.

A straightforward idea is to generate a pattern-coverage *sub-snippet* for each component of D and then merge all sub-snippets. However, different components may contain common patterns. It may be unnecessary to generate and merge sub-snippets for all components to form a pattern-coverage snippet S of D. To improve the compactness of S and the efficiency of its generation, we aim at finding a smallest subset of components that cover all the patterns in D. It is an instance of the well-known *set cover problem* (SCP) where $\mathtt{EDP}(D) \cup \mathtt{LP}(D)$ is the universe and for each component D_j, $\mathtt{EDP}(D_j) \cup \mathtt{LP}(D_j) \subseteq \mathtt{EDP}(D) \cup \mathtt{LP}(D)$ is a set. SCP requires finding the smallest number of sets whose union equals the universe. SCP is NP-hard, and we solve it using a standard greedy algorithm [9].

The extended algorithm `PCSG`, standing for *pattern-coverage snippet generation*, is presented in Algorithm 2. Let $\mathcal{D}$ be the set of all components of D (line 1). P denotes the universe (line 2). Initially S is empty (line 3). Then iteratively until P is fully covered (line 4), we greedily choose a component D_i that contains the largest number of uncovered patterns (line 5). We use `Basic` to generate a pattern-coverage sub-snippet of D_i and add its triples to S (line 6). Finally we update P for the next iteration (line 7).

Moreover, we modify `Basic` to generate a possibly smaller sub-snippet of D_i. Observe that the sub-snippet only needs to cover $(\mathtt{EDP}(D_i) \cup \mathtt{LP}(D_i)) \cap P$ rather than $\texttt{EDP}(D_i) \cup \texttt{LP}(D_i)$. In `Basic`, when formulating GSTP we ignore the groups that correspond to the patterns in $(\texttt{EDP}(D_i) \cup \texttt{LP}(D_i)) \setminus P$.

`PCSG` has the same time complexity as `Basic`.

4.2 Extension to High Heterogeneity: Algorithm PCSG-τ

`PCSG` requires a snippet S to cover all the patterns in D. If D is highly heterogeneous and contains many different patterns, S will inevitably be very large. Below we extend `PCSG` to achieve a trade-off between pattern coverage and snippet size to handle high heterogeneity.

We modify `PCSG` to generate a possibly smaller snippet that only covers the most important patterns in D. Observe that patterns are not equally important. We define the *relative frequency* of an EDP as the proportion of entities that have this EDP in D. The relative frequency of an LP is defined analogously. More frequent patterns are considered more important. We separately rank all EDPs and all LPs in descending order of relative frequency. In `PCSG`, we restrict the universe P to only contain top-ranked EDPs and LPs whose total relative frequency of EDP and total relative frequency of LP exceed τ which is a parameter describing a percentage. The extended algorithm is referred to as `PCSG`-τ.

4.3 Extension to Query Relevance: Algorithm QPCSG(-τ)

Below we extend `PCSG` and `PCSG`-τ to generate *query-biased snippets* to be presented in dataset search to help users determine the relevance of RDF datasets.

Consider a keyword query Q. We modify `PCSG` and `PCSG`-τ to generate a pattern-coverage snippet S that matches all the keywords in Q. Specifically, we view each keyword $q \in Q$ as a *pseudo-pattern*. Each entity or entity link in D is extended to have a set of patterns consisting of: its EDP or LP, and all the pseudo-patterns it matches (computed by an off-the-shelf matcher). Accordingly, when formulating GSTP in `Basic`, for each pseudo-pattern $q \in Q$ we add a group consisting of all entities and entity links that match q. In `PCSG` and `PCSG`-τ, we add all the pseudo-patterns in Q to the universe P, and we refer to the extended algorithms as `QPCSG` and `QPCSG`-τ, respectively.

Regarding the matcher, we adopt the following simple implementation for our experiments. An entity e matches $q \in Q$ if q appears in any triple describing e in D. An entity link $\langle e_i,\ p,\ e_j \rangle$ matches q if q appears in the textual form of p.

5 Evaluation of PCSG(-τ)

We firstly carried out experiments to verify two research hypotheses (RHs) about the *effectiveness* (RH1) and *practicability* (RH2) of our approach `PCSG`(-τ).

- **RH1:** `PCSG`(-τ) generates better snippets than [6, 15].
- **RH2:** `PCSG`(-τ) efficiently generates compact snippets.

All our experiments presented in the paper were serially conducted on an Intel Xeon E7-4820 (2GHz) with 80GB memory for JVM. Source code, experimental data, and example snippets are online: https://github.com/nju-websoft/PCSG.

Table 1: Statistics about RDF datasets.

Portal	#RDF	#triples		#classes		#properties		#EDPs		#LPs	
	datasets	Median	Max	Median	Max	Median	Max	Median	Max	Median	Max
DataHub.io	311	1,272	20,968,879	3	2,030	16	3,982	15	270,224	27	156,722
Data.gov	9,233	4,000	6,343,524	1	2	13	545	3	500	2	1,103
Overall	9,544	4,000	20,968,879	1	2,030	14	3,982	3	270,224	2	156,722

5.1 RDF Datasets

We retrieved all datasets with RDF dumps from two data portals: DataHub.io and Data.gov. We successfully downloaded and used Apache Jena 3.9.0 to parse 9,544 RDF datasets. Their statistics are shown in Table 1. Observe that many entities in datasets from Data.gov are untyped and are described by uniform patterns, probably converted from tabular data.

5.2 Participating Methods

Ours. We implemented three variants: `PCSG`, `PCSG`-90%, `PCSG`-80%.

Baseline. IlluSnip [6, 15] represented the state of the art in snippet generation for RDF datasets. Its original version [6] could not scale to large RDF datasets. We used its anytime version [15] and allowed two hours for computing a snippet. We followed [15] to set its parameters. IlluSnip generated size-bounded snippets containing at most k triples. For a fair comparison, for each RDF dataset we set k to the number of triples in the snippet generated by our approach. Accordingly, there were three variants: IlluSnip, IlluSnip-90%, IlluSnip-80%.

5.3 Experiment 1: Coverage of Schema

`PCSG`(-τ) and IlluSnip both aimed at schema coverage. To verify RH1, we compared their effectiveness in this aspect.

Metrics. We assessed a snippet's capability of covering four kinds of schema-level elements: class, property, EDP, and LP. For each kind, we measured a snippet's *schema coverage rate* by calculating the total relative frequency (defined in Section 4.2) of all the schema-level elements of this kind covered in the snippet. It represented the weighted proportion of covered schema-level elements which were weighted by their numbers of instances. Note that for class and property, their schema coverage rates had been used for evaluating the quality of a snippet in [26] (called `coSkm`). We basically extended it to EDP and LP.

Results. For each approach we calculated its schema coverage rates on each of the 9,544 RDF datasets. The results are summarized in Table 2. All the participating methods achieved (near) perfect schema coverage rates for class and property. It was not surprising: IlluSnip directly optimizes these rates; `PCSG`(-τ) indirectly boosts these rates via pattern coverage. However, IlluSnip was unaware of patterns and exhibited considerably lower schema coverage rates for EDP and

Table 2: Schema coverage rates (mean ± SD).

	Class	Property	EDP	LP
PCSG	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
IlluSnip	0.993 ± 0.079	0.999 ± 0.011	0.822 ± 0.285	0.790 ± 0.320
PCSG-90%	0.999 ± 0.019	0.999 ± 0.006	0.981 ± 0.030	0.976 ± 0.035
IlluSnip-90%	0.991 ± 0.092	0.999 ± 0.013	0.794 ± 0.310	0.762 ± 0.344
PCSG-80%	0.999 ± 0.025	0.998 ± 0.010	0.957 ± 0.061	0.947 ± 0.071
IlluSnip-80%	0.982 ± 0.131	0.998 ± 0.017	0.784 ± 0.317	0.751 ± 0.353

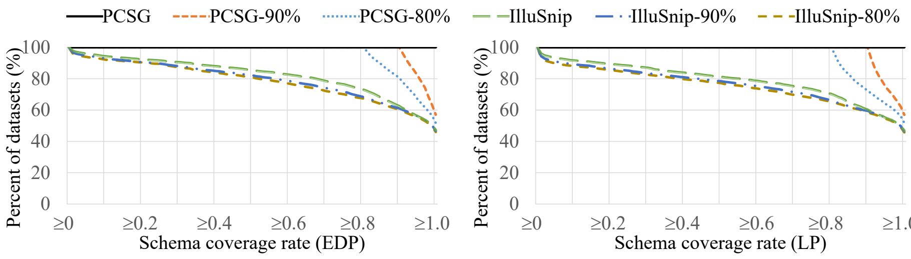


Fig. 2: Cumulative distributions of schema coverage rate.

LP than PCSG(-τ). Indeed, for PCSG-τ these rates were guaranteed to exceed τ, as illustrated by their cumulative distributions over all RDF datasets in Fig. 2. The results supported RH1 in terms of schema coverage.

5.4 Experiment 2: User Preference

Besides schema coverage rates, to verify RH1, we conducted a user study to compare the quality of snippets generated by different methods. We recruited 20 students majoring in computer science from a university via a mailing list, all having the necessary knowledge about RDF and paid to participate.

Procedure and Metrics. We followed the experimental design for snippet comparison described in [6]. Specifically, each participant was randomly assigned ten RDF datasets. For each RDF dataset, the participant could obtain an overview by accessing its metadata and top-twenty most frequent schema-level elements of each kind: class, property, EDP, and LP, each associated with its frequency (i.e., number of instances). Two snippets of this RDF dataset generated by PCSG-80% and IlluSnip-80% were presented in random order to avoid position bias. Each snippet was visualized as a node-link diagram. Its *quality* was to be rated in the range from 1 to 5 indicating how well it exemplified the content of the RDF dataset. The participant was encouraged to briefly explain the rating.

Results. The results of user-rated quality on 200 RDF datasets are summarized in Table 3. Paired two-sample t-test showed that PCSG-80% generated significantly ($p < 0.01$) better snippets than IlluSnip-80%. PCSG-80% was rated ≥ 4 on most RDF datasets (80%), while for IlluSnip-80% this proportion was only 39%, according to the distributions in Fig. 3. On 59% of all RDF datasets PCSG-80% was thought to generate better snippets. Participants in their explanations of

Table 3: User-rated quality.

	Mean ± SD
PCSG-80%	4.09 ± 0.97
IlluSnip-80%	3.24 ± 1.12

	Proportion
PCSG-80% > IlluSnip-80%	59.00%
PCSG-80% = IlluSnip-80%	24.50%
PCSG-80% < IlluSnip-80%	16.50%

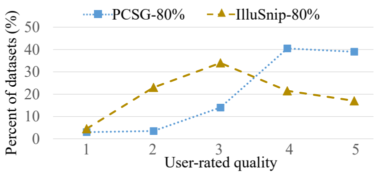


Fig. 3: Distributions of quality.

Table 4: Space savings (mean ± SD).

PCSG	87.02% ± 21.42%
PCSG-90%	89.62% ± 20.98%
PCSG-80%	91.45% ± 19.22%

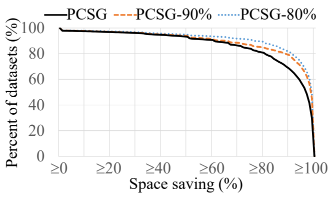


Fig. 4: Cum. distributions of space saving.

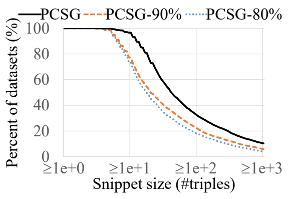


Fig. 5: Cum. distributions of snippet size.

PCSG-80%'s ratings were satisfied with the comprehensive classes and properties included in each entity description which facilitated the comprehension of data content and structure. The results supported RH1 in terms of user preference. However, on 17% of all RDF datasets IlluSnip-80% was thought to generate better snippets. In fact, participants' explanations were mainly concerned about visualization: snippets generated by PCSG-80% were often denser and hence their node-link diagram visualizations appeared more complex. It inspires us to study presentation methods that are more suitable for EDPs and LPs in future work.

5.5 Experiment 3: Space Saving and Run-Time

To verify RH2, we measured the space saving and run-time of our approach.

Metrics. We measured the *space saving* of our approach on an RDF dataset:

$$\text{space saving} = 1 - \frac{\text{number of triples in the generated snippet}}{\text{number of triples in the RDF dataset}}\,, \tag{4}$$

and we reported the *size* of a snippet in terms of the number of triples. Recall that IlluSnip was configured to generate snippets of the same size as ours, thereby having the same space saving and size. We also reported the *run-time* of each approach on an RDF dataset (excluding the time for parsing RDF dumps).

Results. We calculated the space saving of our approach on each of the 9,544 RDF datasets. The results are summarized in Table 4. Our approach substantially reduced the size of an RDF dataset by an average of about 90%. The space savings of PCSG, PCSG-90%, and PCSG-80% were above 95% on 57%, 69%, and 72% of all RDF datasets, respectively, as illustrated by the cumulative distributions in Fig. 4. The median numbers of triples in their generated snippets were only 41, 20, and 17, respectively, as illustrated by the cumulative distributions in Fig 5.

Table 5: Run-time in milliseconds (mean ± SD).

PCSG	2,806 ± 95,310
IlluSnip	856,446 ± 2,103,072
PCSG-90%	1,336 ± 70,896
IlluSnip-90%	572,099 ± 1,722,136
PCSG-80%	981 ± 47,325
IlluSnip-80%	446,110 ± 1,516,651

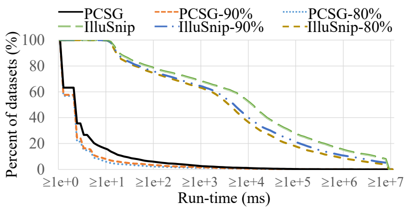


Fig. 6: Cum. distributions of run-time.

These numbers were comparable to the size constraints on compact snippets ($k = 40$ or $k = 20$) proposed in previous research [6, 15, 26]. The results supported the compactness part of RH2. However, we observed a small proportion of snippets containing more than 1,000 triples for highly heterogeneous RDF datasets using diverse combinations of hundreds of properties to describe entities. Their browsing could be facilitated by an interface for filtering as in [17].

For each approach we recorded its run-time on each of the 9,544 RDF datasets. The results are summarized in Table 5. PCSG(-τ) was more than two orders of magnitude faster than IlluSnip. The run-time of PCSG, PCSG-90%, and PCSG-80% was below one second on 98%, 98%, and 99% of all RDF datasets, respectively, as illustrated by the cumulative distributions in Fig. 6. The results supported the efficiency part of RH2. However, for several highly heterogeneous datasets containing thousands of EDPs and LPs, PCSG(-τ) used more than an hour. Though still faster than IlluSnip and acceptable as offline computation, it suggested room for further improving the performance of our approach.

6 Evaluation of QPCSG(-τ)

We also carried out experiments to verify two research hypotheses (RHs) about the *effectiveness* (RH3) and *practicability* (RH4) of our approach QPCSG(-τ).

- **RH3:** QPCSG(-τ) generates better query-biased snippets than [27].
- **RH4:** QPCSG(-τ) efficiently generates compact query-biased snippets.

6.1 Queries and RDF Datasets

From three published research datasets [13, 4, 5] we collected 2,067 keyword queries representing real-world data needs. In our experiments, since keywords were to be matched with the content of an RDF dataset rather than its metadata, we followed an existing annotation scheme [4] to manually annotate each query and removed keywords that were to be matched with metadata (e.g., data format, license). We also removed stop words and filtered out empty queries. For the remaining 1,356 filtered queries, their statistics are shown in Table 6.

To match queries with the 9,544 RDF datasets described in Section 5.1, we employed Apache Lucene 7.5.0 to index the content of each RDF dataset as a

Table 6: Statistics about queries.

Source	#queries	#filtered queries	#words in a filtered query		
			Min	Median	Max
Ref. [13]	449	399	1	3	12
Ref. [4]	1,498	843	1	3	15
Ref. [5]	120	114	3	6	15

Table 7: Statistics about RDF datasets in Q-D pairs.

	#Q-D	#triples		#classes		#properties		#EDPs		#LPs	
	pairs	Median	Max	Median	Max	Median	Max	Median	Max	Median	Max
Overall	13,429	9,049	10,733,302	1	2,030	20	3,982	11	270,224	10	156,722

pseudo document by transforming each triple into a sentence concatenating its subject, predicate, and object in their textual forms (e.g., `rdfs:label`, local name, lexical form). We used the default document retrieval model provided by Lucene and kept ten top-ranked RDF datasets for each query. As a result we obtained 13,429 query-dataset pairs, or Q-D pairs for short. For the RDF datasets in these Q-D pairs, their statistics are shown in Table 7.

6.2 Participating Methods

Ours. We implemented three variants: `QPCSG`, `QPCSG`-90%, `QPCSG`-80%.

Baseline. KSD [27] represented the state of the art in query-biased snippet generation for RDF datasets. We followed [27] to set its parameters. KSD generated size-bounded snippets containing k triples. For a fair comparison, for each RDF dataset we set k to the number of triples in the snippet generated by our approach. Accordingly, there were three variants: KSD, KSD-90%, KSD-80%.

6.3 Experiment 4: Coverage of Query and Schema

`QPCSG`(-τ) and KSD both aimed at query coverage and schema coverage. To verify RH3, we compared their effectiveness in these two aspects.

Metrics. Given a set of keywords in a query, we used two metrics [26] to assess a snippet's capability of covering the query: `coKyw` calculating the proportion of keywords covered in the snippet; `coCnx` calculating the proportion of keyword pairs connected by a path in the RDF graph representation of the snippet. For schema coverage we reused *schema coverage rate* defined in Section 5.3.

Results. For each approach we calculated its `coKyw`, `coCnx`, and schema coverage rates on each of the 13,429 Q-D pairs. The results are summarized in Table 8. All the participating methods achieved satisfying `coKyw` and schema coverage rates for class and property, which was not surprising since they directly or indirectly optimize these metrics. However, `QPCSG`(-τ) achieved noticeably higher `coCnx` than KSD which did not attend to connectivity. KSD was also unaware of patterns and exhibited considerably lower schema coverage rates for EDP and

Table 8: `coKyw`, `coCnx`, and schema coverage rates (mean ± SD).

	`coKyw`	`coCnx`	Class	Property	EDP	LP
`QPCSG`	0.948 ± 0.124	0.841 ± 0.308	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
KSD	0.895 ± 0.239	0.489 ± 0.443	0.944 ± 0.207	0.916 ± 0.219	0.222 ± 0.316	0.070 ± 0.190
`QPCSG`-90%	0.948 ± 0.124	0.839 ± 0.307	0.998 ± 0.013	0.998 ± 0.009	0.947 ± 0.041	0.943 ± 0.042
KSD-90%	0.888 ± 0.248	0.455 ± 0.442	0.905 ± 0.262	0.867 ± 0.261	0.190 ± 0.307	0.047 ± 0.168
`QPCSG`-80%	0.948 ± 0.124	0.836 ± 0.309	0.996 ± 0.020	0.981 ± 0.080	0.890 ± 0.081	0.883 ± 0.082
KSD-80%	0.884 ± 0.253	0.438 ± 0.440	0.892 ± 0.284	0.842 ± 0.281	0.174 ± 0.296	0.037 ± 0.153

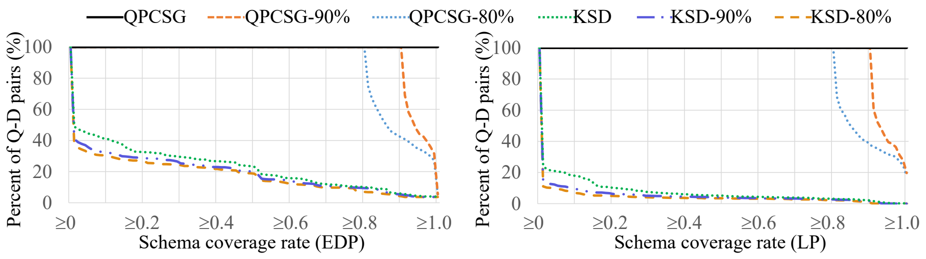


Fig. 7: Cum. distributions of schema coverage rate.

Table 9: Space savings (mean ± SD).

`QPCSG`	88.07% ± 20.78%
`QPCSG`-90%	90.90% ± 19.87%
`QPCSG`-80%	92.38% ± 18.36%

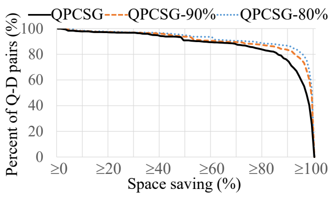


Fig. 8: Cum. distributions of space saving.

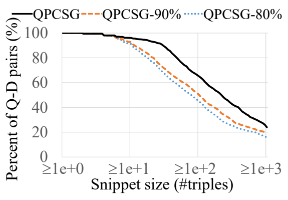


Fig. 9: Cum. distributions of snippet size.

LP than `QPCSG`(-τ). Observe that for `QPCSG`-τ these rates were guaranteed to exceed τ, as illustrated by their cumulative distributions over all Q-D pairs in Fig. 7. The results supported RH3 in terms of query and schema coverage.

6.4 Experiment 5: Space Saving and Run-Time

To verify RH4, we measured the space saving and run-time of our approach.

Metrics. We reused *space saving*, *size*, and *run-time* defined in Section 5.5.

Results. We calculated the space saving of our approach on each of the 13,429 Q-D pairs. The results are summarized in Table 9. Our approach substantially reduced the size of an RDF dataset by an average of about 90%. The space savings of `QPCSG`, `QPCSG`-90%, and `QPCSG`-80% were above 95% on 59%, 76%, and 81% of all Q-D pairs, respectively, as illustrated by the cumulative distributions in Fig. 8. The median numbers of triples in their generated snippets were 215, 101, and 77, respectively, as illustrated by the cumulative distributions in Fig. 9. These numbers appeared larger than those of `PCSG`(-τ) reported in Section 5.5

Table 10: Run-time in milliseconds (mean ± SD).

QPCSG	39,301 ± 268,090
KSD	89,516 ± 718,355
QPCSG-90%	14,215 ± 131,611
KSD-90%	55,458 ± 473,939
QPCSG-80%	13,369 ± 126,801
KSD-80%	44,757 ± 384,312

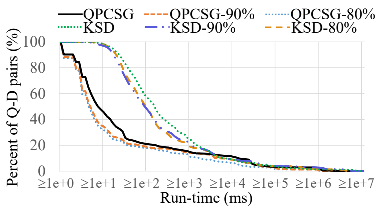


Fig. 10: Cum. distributions of run-time.

because their scopes of statistics were different: the RDF datasets in Q-D pairs here were much larger and more heterogeneous (medians: 9,049 triples, 11 EDPs, 10 LPs in Table 7) than those in Section 5 (medians: 4,000 triples, 3 EDPs, 2 LPs in Table 1) so that many keyword queries were matched with them. However, *on the same RDF dataset*, QPCSG(-τ) did not output noticeably more triples than PCSG(-τ). The compactness part of RH4 was still supported.

For each approach we recorded its run-time on each of the 13,429 Q-D pairs. The results are summarized in Table 10. QPCSG(-τ) was more than twice as fast as KSD. The run-time of QPCSG, QPCSG-90%, and QPCSG-80% was below one second on 85%, 88%, and 91% of all Q-D pairs, though above ten seconds on 11%, 7%, and 4%, respectively, as illustrated by the cumulative distributions in Fig. 10. Again, QPCSG(-τ) actually did not use noticeably more time than PCSG(-τ) *on the same RDF dataset*. The efficiency part of RH4 was also supported.

7 Empirical Comparison with Summary

Since PCSG(-τ) injects the strength of summary (i.e., pattern) into snippet, it would be desirable to empirically compare them. We conducted a user study to compare their usefulness for performing the task of SPARQL query completion [24]. We recruited 30 students majoring in computer science from a university, all having the necessary knowledge about SPARQL and paid to participate.

RDF Dataset. For this experiment we used DBpedia 2016-10.

Participating Methods. We compared PCSG-80% with ABSTAT [24], a popular summarization method. We used ABSTAT to compute a summary containing all the ontologically minimal patterns of triples in DBpedia, and we reproduced its tabular interface for presenting and filtering patterns with autocomplete [17]. For a fair comparison, we implemented a similar interface for PCSG-80%: entity descriptions in the snippet were grouped by EDP and then were presented in a tabular interface for presenting and filtering EDPs with autocomplete.

Procedure and Metrics. We followed the experimental design for SPARQL query completion described in [24]. Firstly, each participant performed two *simple tasks*: one using the snippet generated by PCSG-80% and the other using the summary computed by ABSTAT, performed in random order to avoid position bias. Each simple task asked the participant to rely on PCSG-80% or ABSTAT to

Table 11: Accuracy and time (mean ± SD) for completing a SPARQL query.

	Accuracy			Time (seconds)		
	Simple Task	Complex Task	Overall	Simple Task	Complex Task	Overall
PCSG-80%	0.900 ± 0.300	0.900 ± 0.300	0.900 ± 0.300	85.9 ± 39.0	164.0 ± 84.6	124.9 ± 76.6
ABSTAT	0.933 ± 0.249	0.833 ± 0.373	0.883 ± 0.321	117.6 ± 51.1	214.3 ± 154.1	166.0 ± 124.6

Table 12: User-rated usefulness.

	Mean ± SD
PCSG-80%	4.47 ± 0.62
ABSTAT	3.60 ± 0.95

	Proportion
PCSG-80% > ABSTAT	66.67%
PCSG-80% = ABSTAT	26.67%
PCSG-80% < ABSTAT	6.67%

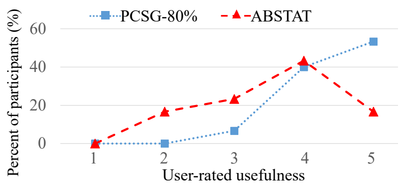


Fig. 11: Distributions of usefulness.

convert a natural language query into a SPARQL query consisting of two triple patterns. The participant was given an incomplete SPARQL query where a predicate was left blank to type in a property. Next, in the same way the participant performed two *complex tasks*: completing a SPARQL query consisting of four triple patterns where two predicates and one object were left blank to type in two properties and one class, respectively. Before all tasks the participant was given a tutorial on the two interfaces with a warm-up task. After all tasks the participant rated each method in the range from 1 to 5 indicating its *usefulness*. The participant was encouraged to briefly explain the rating. We also reported the binary *accuracy* of a completed SPARQL query by comparing it with the gold standard, and reported the *time* for completing a SPARQL query.

Results. The results of accuracy and time for completing 120 SPARQL queries are summarized in Table 11. While paired two-sample t-test showed that the difference between the accuracy using PCSG-80% and the accuracy using ABSTAT was not statistically significant ($p > 0.05$), participants spent 25% less time using PCSG-80% and this difference was statistically significant ($p < 0.01$). The results of user-rated usefulness are summarized in Table 12. Paired two-sample t-test showed that PCSG-80% was significantly ($p < 0.01$) more useful than ABSTAT. PCSG-80% was rated ≥ 4 by most participants (93%), while for ABSTAT this proportion was 60%, according to the distributions in Fig. 11. By 66.67% of all participants PCSG-80% was thought to be more useful than ABSTAT. Participants in their explanations of ratings preferred PCSG-80% because it helped participants find all the needed classes and properties in one entity description, while using ABSTAT they had to find multiple patterns of triples. Besides, compared with abstract patterns in the summary, concrete entity descriptions in the snippet exemplified the use of classes and properties and improved participants' confidence. However, by 6.67% of all participants ABSTAT was thought to be more useful than PCSG-80%. The explanations were concerned about complexity: participants were overloaded by some large entity descriptions in the snippet.

8 Conclusion and Future Work

We presented novel methods PCSG(-τ) and QPCSG(-τ) for generating pattern-coverage snippets for RDF datasets. They effectively generated better snippets than existing methods in terms of schema coverage, query coverage, and user preference, and their space savings and run-time demonstrated practicability. In the future, we plan to optimize PCSG(-τ) and QPCSG(-τ) to address some shortcomings observed in the experiments. Firstly, we observed a few large snippets even with $\tau = 80\%$. To solve this problem, we plan to adapt our approach to generating a size-bounded snippet that covers the most frequent patterns in an RDF dataset. We will also consider merging similar EDPs [28] or mining common sub-EDPs to reduce snippet size. However, such a snippet may not precisely reflect how entities are described in the dataset. Secondly, to address participants' concerns about visualization in the user study, we will investigate presentation methods [19] that are more suitable for showing patterns.

Combining the strengths of snippet and summary, our snippet appeared more useful than the summary computed by ABSTAT in assisting SPARQL query completion over DBpedia. However, based on a single downstream task and a comparison with a single method, one shall not draw a general conclusion that our snippets can be substituted for all summaries in all tasks. Rather, we believe they are complementary. In future work we will carry out experiments to comprehensively evaluate various summarization and snippet generation methods.

Acknowledgements

This work was supported by the NSFC (62072224).

References

1. Campinas, S., Delbru, R., Tummarello, G.: Efficiency and precision trade-offs in graph summary algorithms. In: IDEAS 2013. pp. 38–47 (2013)
2. Cebiric, S., Goasdoué, F., Kondylakis, H., Kotzinos, D., Manolescu, I., Troullinou, G., Zneika, M.: Summarizing semantic graphs: a survey. VLDB J. **28**(3) (2019)
3. Chapman, A., Simperl, E., Koesten, L., Konstantinidis, G., Ibáñez, L., Kacprzak, E., Groth, P.: Dataset search: a survey. VLDB J. **29**(1), 251–272 (2020)
4. Chen, J., Wang, X., Cheng, G., Kharlamov, E., Qu, Y.: Towards more usable dataset search: From query characterization to snippet generation. In: CIKM 2019. pp. 2445–2448 (2019)
5. Chen, Z., Jia, H., Heflin, J., Davison, B.D.: Leveraging schema labels to enhance dataset search. In: ECIR 2020. pp. 267–280 (2020)
6. Cheng, G., Jin, C., Ding, W., Xu, D., Qu, Y.: Generating illustrative snippets for open data on the web. In: WSDM 2017. pp. 151–159 (2017)
7. Cheng, G., Jin, C., Qu, Y.: HIEDS: A generic and efficient approach to hierarchical dataset summarization. In: IJCAI 2016. pp. 3705–3711 (2016)
8. Ellefi, M.B., Bellahsene, Z., Breslin, J.G., Demidova, E., Dietze, S., Szymanski, J., Todorov, K.: RDF dataset profiling - a survey of features, methods, vocabularies and applications. Semant. Web **9**(5), 677–705 (2018)

9. Feige, U.: A threshold of ln n for approximating set cover. J. ACM **45**(4), 634–652 (1998)
10. Fokoue, A., Meneguzzi, F., Sensoy, M., Pan, J.Z.: Querying linked ontological data through distributed summarization. In: AAAI 2012 (2012)
11. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K., Umbrich, J.: Data summaries for on-demand queries over linked data. In: WWW 2010 (2010)
12. Heling, L., Acosta, M.: Estimating characteristic sets for RDF dataset profiles based on sampling. In: ESWC 2020. pp. 157–175 (2020)
13. Kacprzak, E., Koesten, L., Tennison, J., Simperl, E.: Characterising dataset search queries. In: WWW 2018. pp. 1485–1488 (2018)
14. Khatchadourian, S., Consens, M.P.: ExpLOD: summary-based exploration of interlinking and RDF usage in the linked open data cloud. In: ESWC 2010, Part II. pp. 272–287 (2010)
15. Liu, D., Cheng, G., Liu, Q., Qu, Y.: Fast and practical snippet generation for RDF datasets. ACM Trans. Web **13**(4), 19:1–19:38 (2019)
16. Liu, Q., Cheng, G., Gunaratna, K., Qu, Y.: Entity summarization: State of the art and future challenges. CoRR **abs/1910.08252** (2019)
17. Palmonari, M., Rula, A., Porrini, R., Maurino, A., Spahiu, B., Ferme, V.: ABSTAT: linked data summaries with abstraction and statistics. In: ESWC 2015 Satellite Events. pp. 128–132 (2015)
18. Pan, J.Z.: Resource description framework. In: Handbook on Ontologies, pp. 71–90. Springer (2009)
19. Parvizi, A., Mellish, C., van Deemter, K., Ren, Y., Pan, J.Z.: Selecting ontology entailments for presentation to users. In: KEOD 2014. pp. 382–387 (2014)
20. Rietveld, L., Hoekstra, R., Schlobach, S., Guéret, C.: Structural properties as proxy for semantic relevance in RDF graph sampling. In: ISWC 2014. pp. 81–96 (2014)
21. Safavi, T., Belth, C., Faber, L., Mottin, D., Müller, E., Koutra, D.: Personalized knowledge graph summarization: From the cloud to your pocket. In: ICDM 2019. pp. 528–537 (2019)
22. Shi, Y., Cheng, G., Kharlamov, E.: Keyword search over knowledge graphs via static and dynamic hub labelings. In: WWW 2020. pp. 235–245 (2020)
23. Song, Q., Wu, Y., Lin, P., Dong, X., Sun, H.: Mining summaries for knowledge graph search. IEEE Trans. Knowl. Data Eng. **30**(10), 1887–1900 (2018)
24. Spahiu, B., Porrini, R., Palmonari, M., Rula, A., Maurino, A.: ABSTAT: ontology-driven linked data summaries with pattern minimalization. In: ESWC 2016 Satellite Events. pp. 381–395 (2016)
25. Wang, K., Wang, Z., Topor, R.W., Pan, J.Z., Antoniou, G.: Eliminating concepts and roles from ontologies in expressive descriptive logics. Comput. Intell. **30**(2), 205–232 (2014)
26. Wang, X., Chen, J., Li, S., Cheng, G., Pan, J.Z., Kharlamov, E., Qu, Y.: A framework for evaluating snippet generation for dataset search. In: ISWC 2019, Part I. pp. 680–697 (2019)
27. Wang, X., Cheng, G., Kharlamov, E.: Towards multi-facet snippets for dataset search. In: PROFLILES & SemEx 2019. pp. 1–6 (2019)
28. Zneika, M., Lucchese, C., Vodislav, D., Kotzinos, D.: Summarizing linked data RDF graphs using approximate graph pattern mining. In: EDBT 2016. pp. 684–685 (2016)
29. Zneika, M., Vodislav, D., Kotzinos, D.: Quality metrics for RDF graph summarization. Semant. Web **10**(3), 555–584 (2019)